FOODiE, Fortran Object oriented Ordinary Differential Equations integration library based on Abstract Calculus Pattern

Zaghi S.[a,1,*], Curcic M.[b,2], Rouson D.[c,3], Beekman I.[c,4]

[a]*CNR-INSEAN, Istituto Nazionale per Studi ed Esperienze di Architettura Navale, Via di Vallerano 139, Rome, Italy, 00128*
[b]*Ocean Sciences Rosenstiel School of Marine and Atmospheric Science, University of Miami, 4600 Rickenbacker Causeway Miami, FL 33149-1098 +1 305.421.4000*
[c]*Sourcery Institute 482 Michigan Ave., Berkeley, CA 94707*

Abstract

To be written.

Keywords: Ordinary Differential Equations (ODE), Partial Differential Equations (PDE), Object Oriented Programming (OOP), Abstract Calculus Pattern (ACP), Fortran,

PROGRAM SUMMARY

Manuscript Title: FOODiE, Fortran Object oriented Ordinary Differential Equations integration library based on Abstract Calculus Pattern
Authors: Zaghi, S., Curcic, M., Rouson, D., Beekman, I.
Program title: FOODiE
Journal Reference:
Catalogue identifier:
Licensing provisions: GNU General Public License (GPL) v3
Programming language: Fortran (standard 2008 or newer); developed and tested with GNU gfortran 5.2 or newer
Computer(s) for which the program has been designed: designed for shared-memory multi-cores workstations and for hybrid distributed/shared-memory supercomputers, but any computer system with a Fortran (2008+) compiler is suited
Operating system(s) for which the program has been designed: designed for POSIX architecture and tested on GNU/Linux one
RAM required to execute with typical data: bytes: $[1MB, 1GB] \times core$, simulation-dependent
Has the code been vectorised or parallelized?: the library is not aware of the parallel back-end, it providing a high-level models, but the provided tests suite shows parallel usage by means of MPI library and OpenMP paradigm
Number of processors used: tested up to 256
Supplementary material:
Keywords: ODE, PDE, OOP, ACP, Fortran
CPC Library Classification: 4.3 Differential Equations, 4.10 Interpolation, 12 Gases and Fluids
External routines/libraries used:
CPC Program Library subprograms used:
Nature of problem:
Numerical integration of (general) Ordinary Differential Equations system
Solution method:

*Corresponding author
Email addresses: `stefano.zaghi@cnr.it` (Zaghi S.), `milan@orca.rsmas.miami.edu` (Curcic M.), `damian@sourceryinstitute.org` (Rouson D.), `izaak@izaakbeekman.com` (Beekman I.)
[1]Ph. D., Aerospace Engineer, Research Scientist, Dept. of Computational Hydrodynamics at CNR-INSEAN.
[2]Ph.D. Meteorology and Physical Oceanography, Research Scientist, Dept. of Ocean Sciences Rosenstiel School of Marine and Atmospheric Science at University of Miami
[3]Ph.D. Mechanical Engineering, Founder and President Sourcery Institute and Sourcery, Inc.
[4]Graduate Research Assistant, Princeton/UMD CCROCCO LAB

Restrictions:
Unusual features:
Additional comments:
Running time:
References:

1. Introduction

1.1. Background

Initial Value Problem (IVP, or Cauchy problem) constitutes a class of mathematical models of paramount relevance, it being applied to the modelling of a wide range of problems. Briefly, an IVP is an Ordinary Differential Equations system (ODE) coupled with specified initial values of the unknown state variables, the solution of which are searched at a given time after the initial time considered.

The prototype of IVP can be expressed as:

$$\begin{aligned} U_t &= R(t, U) \\ U_0 &= U(t_0) \end{aligned} \tag{1}$$

where $U(t)$ is the vector of state variables being a function of the time-like independent variable t, $U_t = \frac{dU}{dt} = R(t, U)$ is the (vectorial) residual function and $U(t_0)$ is the (vectorial) initial conditions, namely the state variables function evaluated at the initial time t_0. In general, the residual function R is a function of the state variable U through which it is a function of time, but it can be also a direct function of time, thus in general $R = R(t, U(t))$ holds.

The problem prototype 1 is ubiquitous in the mathematical modelling of physical problems: essentially whenever an *evolutionary* phenomenon is considered the prevision (simulation) of the future solutions involves the solution of an IVP. As a matter of facts, many physical problems (fluid dynamics, chemistry, biology, evolutionary-anthropology, etc...) are described by means of an IVP.

It is worth noting that the state vector variables U and its corresponding residual function $U_t = \frac{dU}{dt} = R(t, U)$ are *problem dependent*: the number and the meaning of the state variables as well as the equations governing their evolution (that are embedded into the residual function) are different for the Navier-Stokes conservation laws with respect the Burgers one, as an example. Nevertheless, the *solvers* used for the prediction of the Navier-Stokes equations evolution (hereafter the *solution*) are the same that are used for Burgers equations time-integration. As a consequence, the solution of the IVP model prototype can be generalized, allowing the application of the same solver to many different problems, thus eliminating the necessity to re-implement the same solver for each different problem.

FOODiE library is designed for solving the generalized IVP 1, it being completely unaware of the actual problem's definition. FOODiE library provides a high-level, well-documented, simple Application Program Interface (API) for many well-known ODE integration solvers, its aims being twofold:

- provide a robust set of ODE solvers ready to be applied to a wide range of different problems;
- provide a simple framework for the rapid development of new ODE solvers.

Add citations to IVP, Cauchy, ODE references.

1.2. Related Works

To be written.

1.3. Motivations and Aims

FOODiE library must:

- be written in modern Fortran (standard 2008 or newer);
- be written by means of Object Oriented Programming (OOP) paradigm;

- be well documented;
- be Tests Driven Developed (TDD);
- be collaboratively developed;
- be free.

FOODiE, meaning Fortran Object oriented Ordinary Differential Equations integration library, has been developed with the aim to satisfy the above specifications. The present paper is its first comprehensive presentation.

The Fortran (Formula Translator) programming language is the *de facto* standard into computer science field: it strongly facilitates the effective and efficient translation of (even complex) mathematical and numerical models into an operative software without compromise on computations speed and accuracy. Moreover, its simple syntax is suitable for scientific researchers that are interested (and skilled) in the physical aspects of the numerical computations rather than computer technicians. Consequently, we develop FOODiE using Fortran language: FOODiE is written by research scientists for research scientists.

One key-point of the FOODiE development is the *problem generalization*: the problem solved must be the IVP 1 rather than any of its actual definitions. Consequently, we must rely on a *generic* implementation of the solvers. To this aim, OOP is very useful: it allows to express IVP 1 in a very concise and clear formulation that is really generic. In particular, our implementation is based on *Abstract Calculus Pattern* (ACP) concept.

The Abstract Calculus Pattern. The abstract calculus pattern provides a simple solution for the connection between the very high-level expression of IVP 1 and the eventual concrete (low-level) implementation of the ODE problem being solved. ACP essentially constitutes a *contract* based on an Abstract Data Type (ADT): we specify an ADT supporting a certain set of mathematical operators (differential and integral ones) and implement FOODiE solvers only on the basis of this ADT. FOODiE clients must formulate the ODE problem under integration defining their own concrete extensions of our ADT (implementing all the deferred operators). Such an approach defines the abstract calculus pattern: FOODiE solvers are aware of only the ADT, while FOODiE clients extend the ADT for defining the concrete ODE problem.

Is is worth noting that this ACP emancipates the solvers implementations from any low-level problem-dependent details: the ODE solvers developed with this pattern are extremely concise, clear, maintainable and less errors-prone with respect a low-level (non abstract) pattern. Moreover, the FOODiE clients can use solvers being extremely robust: as a matter of facts, FOODiE solvers are expressed in a formulation very close to the mathematical one and are tested on an extremely varying family of problems. As shown in the following, such a great flexibility does not compromise the computational efficiency.

The present paper is organized as following: in section 2 a brief description of the mathematical and numerical methods currently implemented into FOODiE is presented; in section 3 a detailed discussion on the implementation specifications is provided by means of an analytical code-listings review; in section 4 a verification analysis on the results of FOODiE applications is presented; finally, in section 5 concluding remarks and perspectives are depicted.

Add citations to Fortran standards, OOP, TDD, free software, ACP, ADT, Rouson's book references.

2. Mathematical and Numerical Models

In many (most) circumstances, the solution of equation 1 cannot be computed in a closed, exact form (even if it exists and is unique) due to the complexity and nature of the residual functions, that is often non linear. Consequently, the problem is often solved relying on a numerical approach: the solution of system 1 at a time t^n, namely $U(t^n)$, is approximated by a subsequent time-marching approximations $U_0 = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_N \approx U(t^n)$ where the relation $u_i \rightarrow u_{i+1}$ implies a *stepping, numerical integration* from the time t^i to time t^{i+1} and N is the total number of numerical time steps necessary to evolve the initial conditions toward the searched solution $U(t^n)$. To this aim, many numerical schemes have been devised. Notably, the numerical schemes of practical usefulness must posses some necessary proprieties such as *consistency* and *stability* to ensure that the numerical approximation *converges* to the exact solution as the numerical time step tends to zero. A detailed discussion of these details is out the scope of the present work and is omitted. Here, we briefly recall some classifications necessary to introduce the schemes implemented into the FOODiE library.

A non comprehensive classification of the most widely used schemes could distinguish between *multi-step* versus *one-step* schemes and between *explicit* versus *implicit* schemes.

Essentially, the multi-step schemes have been developed to obtain an accurate approximation of the subsequent numerical steps using the informations contained into the previously computed steps, thus this approach relates the next step approximation to a set of the previously computed steps. On the contrary, a one-step scheme evolves the solution toward the next step using only the information coming from the current time approximation. In the framework of one-step schemes family an equivalent accurate approximation can be obtained by means of a multi-stage approach as the one due to Runge-Kutta. The current version of FOODiE provides schemes belonging to both these families.

The other ODE solvers classification concerns with explicit or implicit nature of the schemes employed. Briefly, an explicit scheme computes the next step approximation using the previously computed steps at most to the current time, whereas an implicit scheme uses also the next step approximation (that is the unknown), thus it requires extra computations. The implicit approach is of practical use for *stiff* systems where the usage of explicit schemes could require an extremely small time step to evolve in a *stable* way the solution. Currently, FOODiE provides only explicit solvers.

FOODiE currently implements the following ODE schemes:

- one-step schemes:
  - explicit forward Euler scheme, it being 1^{st} order accurate;
  - explicit Runge-Kutta schemes:
    - low storage Runge-Kutta schemes:
      - 5-stages 2N registers schemes, it being 4^{th} order accurate;
    - TVD/SSP Runge-Kutta schemes:
      - 2-stages, it being 2^{nd} order accurate;
      - 3-stages, it being 3^{rd} order accurate;
      - 5-stages, it being 4^{th} order accurate;
- multi-step schemes:
  - Adams-Bashforth schemes:
    - 2-steps, it being 2^{nd} order accurate;
    - 3-steps, it being 3^{rd} order accurate;
  - Leapfrog schemes:
    - 2-steps unfiltered, it being 2^{nd} order accurate, but mostly *unstable*;
    - 2-steps Robert-Asselin filtered, it being 1^{st} order accurate;
    - 2-steps Robert-Asselin-Williams filtered, it being 3^{rd} order accurate;

Add citations to all solvers reference papers.

2.1. *The explicit forward Euler scheme*

To be written.

2.2. *The explicit low storage Runge-Kutta class of schemes*

To be written.

2.3. *The explicit TVD/SSP Runge-Kutta class of schemes*

To be written.

2.4. *The explicit Adams-Bashforth class of schemes*

To be written.

2.5. *The leapfrog solver*

To be written.

3. Application Program Interface

In this section we review the FOODiE API providing a detailed discussion of the implementation choices.

3.1. *The main FOODiE Abstract Data Type: the integrand type*

Our ACP is based on one main ADT, the *integrand* type, the definition of which is shown in listing 1.

```
type, abstract :: integrand
  !< Abstract type for building FOODiE ODE integrators.
  contains
    ! public deferred procedures that concrete integrand-field must implement
    procedure(time_derivative),       pass(self), deferred, public :: t
    procedure(update_previous_steps), pass(self), deferred, public :: update_previous_steps
    procedure(previous_step),         pass(self), deferred, public :: previous_step
    ! operators
    procedure(symmetric_operator),   pass(lhs), deferred, public :: integrand_multiply_integrand
    procedure(integrand_op_real),    pass(lhs), deferred, public :: integrand_multiply_real
    procedure(real_op_integrand),    pass(rhs), deferred, public :: real_multiply_integrand
    procedure(symmetric_operator),   pass(lhs), deferred, public :: add
    procedure(symmetric_operator),   pass(lhs), deferred, public :: sub
    procedure(assignment_integrand), pass(lhs), deferred, public :: assign_integrand
    ! operators overloading
    generic, public :: operator(+) => add
    generic, public :: operator(-) => sub
    generic, public :: operator(*) => integrand_multiply_integrand, &
                                      real_multiply_integrand, &
                                      integrand_multiply_real
    generic, public :: assignment(=) => assign_integrand
endtype integrand
```

Listing 1: integrand type definition

The *integrand* type does not implement any actual integrand field, it being and abstract type. It only specifies which deferred procedures are necessary for implementing an actual concrete integrand type that can use a FOODiE solver. Listing 1 specifies the deferred type bound procedures that clients must implement into their own concrete extension of the *integrand* ADT. These procedures are analyzed in the following paragraphs.

3.1.1. *Time derivative procedure, the residual function*

The abstract interface of the time derivative procedure *t* is shown in listing 2.

```
function time_derivative(self, n, t) result(dState_dt)
import :: integrand, R_P, I_P
class(integrand),     intent(IN) :: self       !< Integrand field.
integer(I_P), optional, intent(IN) :: n        !< Time level.
real(R_P),    optional, intent(IN) :: t        !< Time.
class(integrand), allocatable    :: dState_dt  !< Result of the time derivative function of integrand field.
endfunction time_derivative
```

Listing 2: time derivative procedure interface

This procedure-function takes three arguments, the first passed as a *type bounded* argument, while the latter are optional, and returns an integrand object. The passed dummy argument, *self*, is a polymorphic argument that could be any extensions of the *integrand* ADT. The optional argument *n* indicated the *time level* at which the residual function must be evaluated: this argument is necessary for multi-level (often referred as multi-step) solvers such as the Adams-Bashforth or leapfrog ones, but it can be omitted for other classes of solvers such the one-step multi-stages family. The optional argument *t* is the time at which the residual function must be computed: it can be omitted in the case the residual function does not depend directly from time.

Commonly, into the concrete implementation of this deferred abstract procedure clients embed the actual ODE equations being solved. As an example, for the Burgers equation, that is a Partial Differential Equations (PDE) system involving also a boundary value problem, this procedure embeds the spatial operator that convert the PDE to a system of algebraic ODE. As a consequence, the eventual concrete implementation of this procedure can be very complex and errors-prone. Nevertheless, the FOODiE solvers are implemented only on the above abstract interface, thus emancipating the solvers implementation from any concrete complexity.

Add citations to Burgers, Adams-Bashfort, leapfrog references.

3.1.2. *Update previous time steps procedure*

The abstract interface of the *update previous time steps* procedure is shown in listing 3.

```fortran
subroutine update_previous_steps(self, filter, weights)
import :: integrand, R_P
class(integrand),           intent(INOUT) :: self       !< Integrand field.
class(integrand), optional, intent(IN)    :: filter     !< Filter field displacement.
real(R_P),        optional, intent(IN)    :: weights(:) !< Weights for filtering the steps.
endsubroutine update_previous_steps
```

Listing 3: update previous time steps procedure interface

To be completed.

3.1.3. *Previous step procedure*

The abstract interface of the *previous step* procedure is shown in listing 4.

```fortran
function previous_step(self, n) result(previous)
import :: integrand, I_P
class(integrand), intent(IN)  :: self     !< Integrand field.
integer(I_P),     intent(IN)  :: n        !< Time level.
class(integrand), allocatable :: previous !< Selected previous time integrand field.
endfunction previous_step
```

Listing 4: previous step procedure interface

To be completed.

3.1.4. *Symmetric operators procedures*

The abstract interface of *symmetric* procedures is shown in listing 5.

```fortran
function symmetric_operator(lhs, rhs) result(operator_result)
import :: integrand
class(integrand), intent(IN)  :: lhs             !< Left hand side.
class(integrand), intent(IN)  :: rhs             !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction symmetric_operator
```

Listing 5: symmetric operator procedure interface

To be completed.

3.1.5. *Integrand/real and real/integrand operators procedures*

The abstract interfaces of *Integrand/real and real/integrand operators* procedures are shown in listing 6.

```fortran
function integrand_op_real(lhs, rhs) result(operator_result)
import :: integrand, R_P
class(integrand), intent(IN)  :: lhs             !< Left hand side.
real(R_P),        intent(IN)  :: rhs             !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction integrand_op_real

function real_op_integrand(lhs, rhs) result(operator_result)
import :: integrand, R_P
real(R_P),        intent(IN)  :: lhs             !< Left hand side.
class(integrand), intent(IN)  :: rhs             !< Right hand side.
class(integrand), allocatable :: operator_result !< Operator result.
endfunction real_op_integrand
```

Listing 6: Integrand/real and real/integrand operators procedure interfaces

To be completed.

3.1.6. Integrand assignment procedure

The abstract interface of *integrand assignment* procedure is shown in listing 7.

```
subroutine assignment_integrand(lhs, rhs)
import :: integrand
class(integrand), intent(INOUT) :: lhs !< Left hand side.
class(integrand), intent(IN)    :: rhs !< Right hand side.
endsubroutine assignment_integrand
```

Listing 7: integrand assignment procedure interface

To be completed.

3.2. *The explicit forward Euler solver*

To be written.

3.3. *The explicit Adams-Bashforth class of solvers*

To be written.

3.4. *The leapfrog solver*

To be written.

3.5. *The explicit low storage Runge-Kutta class of solvers*

To be written.

3.6. *The explicit TVD/SSP Runge-Kutta class of solvers*

To be written.

4. Tests and Examples

To be written.

4.1. *Oscillation equations test*

To be completed.

$$\begin{gathered}U_t = R(U)\\ U = \begin{bmatrix} x \\ y \end{bmatrix} \quad R(U) = \begin{bmatrix} -fy \\ fx \end{bmatrix}\end{gathered} \tag{2}$$

where the frequency is chosen as $f = 10^4$. The ODE system 2 is completed by the following initial conditions:

$$\begin{gathered}x(t_0) = 0\\ y(t_0) = 1\end{gathered} \tag{3}$$

where $t_0 = 0$ is the initial time considered. The exact solution is:

$$\begin{gathered}x(t_0 + \Delta t) = X_0 cos(f\Delta t) - y_0 sin(f\Delta t)\\ y(t_0 + \Delta t) = X_0 sin(f\Delta t) + y_0 cos(f\Delta t)\end{gathered} \tag{4}$$

where Δt is an arbitrary time step.

4.1.1. *Errors Analysis*

For the analysis of the accuracy of each solvers implemented into FOODiE library, we have integrated the Oscillation equations 2 with different, decreasing time steps in the range $[5000, 2500, 1250, 625, 320, 100]$.

The error is estimated by the L2 norm of the difference between the exact (U_e) and the numerical ($U_{\Delta t}$) solutions for each time step:

$$\varepsilon(\Delta t) = ||U_e - U_{\Delta t}||_2 = \sqrt{\sum_{s=1}^{N_s} (U_e(t_0 + s * \Delta t) - U_{\Delta t}(t_0 + s * \Delta t))^2} \tag{5}$$

Using two pairs of subsequent-decreasing time steps solution is possible to estimate the order of accuracy of the solver employed computing the *observed order* of accuracy:

$$p = \frac{log10\left(\frac{\varepsilon(\Delta t_1)}{\varepsilon(\Delta t_2)}\right)}{log10\left(\frac{\Delta t_1}{\Delta t_2}\right)} \tag{6}$$

where $\frac{\Delta t_1}{\Delta t_2} > 1$.

Table 1 summarizes the errors analysis.

Adams-Bashforth.

Leapfrog.

Low Storage Runge-Kutta.

TVD/SSP Runge-Kutta.

Table 1: Oscillation test errors analysis

Solver	Time Step	Error X	Error Y	Order X	Order Y
Adams-Bashforth 2 steps	5000.0	0.109E+04	0.112E+04	/	/
	2500.0	0.246E+02	0.243E+02	5.47	5.53
	1250.0	0.783E+01	0.789E+01	1.65	1.62
	625.0	0.268E+01	0.272E+01	1.55	1.54
	320.0	0.976E+00	0.990E+00	1.51	1.51
	100.0	0.171E+00	0.173E+00	1.50	1.50
Adams-Bashforth 3 steps	5000.0	0.816E+01	0.809E+01	/	/
	2500.0	0.266E+01	0.264E+01	1.62	1.62
	1250.0	0.120E+01	0.120E+01	1.15	1.14
	625.0	0.545E+00	0.543E+00	1.14	1.14
	320.0	0.223E+00	0.222E+00	1.34	1.34
	100.0	0.418E-01	0.416E-01	1.44	1.44
leapfrog 2 steps	5000.0	0.151E+02	0.152E+02	/	/
	2500.0	0.500E+01	0.504E+01	1.60	1.59
	1250.0	0.190E+01	0.193E+01	1.40	1.39
	625.0	0.181E+01	0.181E+01	0.07	0.09
	320.0	0.157E+01	0.157E+01	0.21	0.21
	100.0	0.100E+01	0.100E+01	0.39	0.39
low storage Runge-Kutta 5 stages	5000.0	0.120E+00	0.122E+00	/	/
	2500.0	0.106E-01	0.107E-01	3.51	3.51
	1250.0	0.935E-03	0.947E-03	3.50	3.50
	625.0	0.826E-04	0.836E-04	3.50	3.50
	320.0	0.793E-05	0.803E-05	3.50	3.50
	100.0	0.135E-06	0.137E-06	3.50	3.50
TVD/SSP Runge-Kutta 2 stages	5000.0	0.316E+02	0.319E+02	/	/
	2500.0	0.892E+01	0.894E+01	1.83	1.84
	1250.0	0.301E+01	0.305E+01	1.57	1.55
	625.0	0.106E+01	0.107E+01	1.51	1.51
	320.0	0.387E+00	0.392E+00	1.50	1.50
	100.0	0.676E-01	0.685E-01	1.50	1.50
TVD/SSP Runge-Kutta 3 stages	5000.0	0.255E+01	0.252E+01	/	/
	2500.0	0.523E+00	0.516E+00	2.28	2.29
	1250.0	0.944E-01	0.931E-01	2.47	2.47
	625.0	0.167E-01	0.165E-01	2.50	2.50
	320.0	0.314E-02	0.310E-02	2.50	2.50
	100.0	0.171E-03	0.169E-03	2.50	2.50
TVD/SSP Runge-Kutta 5 stages	5000.0	0.139E+00	0.141E+00	/	/
	2500.0	0.122E-01	0.124E-01	3.50	3.50
	1250.0	0.108E-02	0.110E-02	3.50	3.50
	625.0	0.956E-04	0.969E-04	3.50	3.50
	320.0	0.937E-05	0.949E-05	3.47	3.47
	100.0	0.512E-06	0.519E-06	2.50	2.50

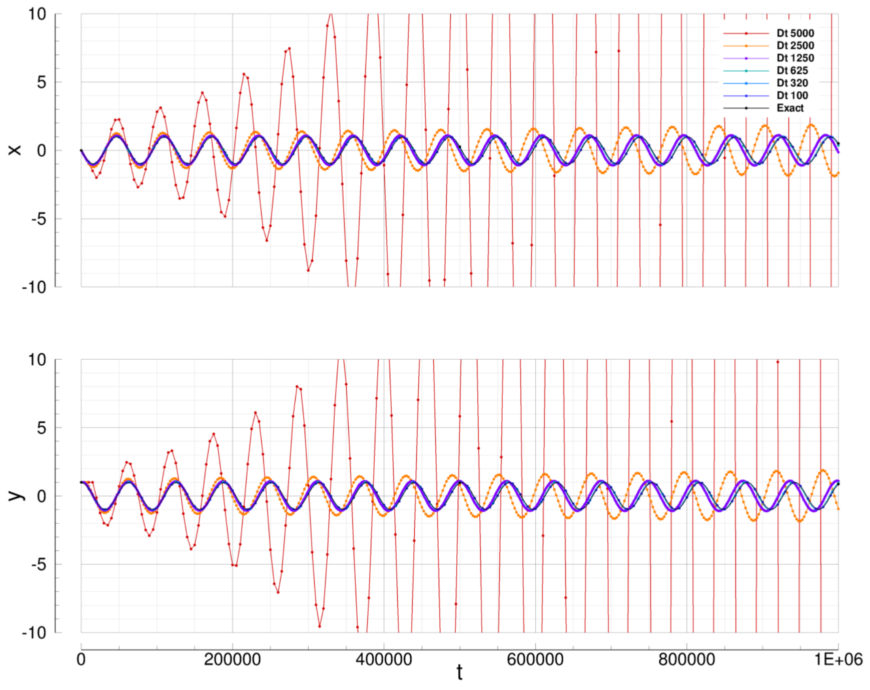

Figure 1: Oscillation equations solutions computed by means of Adams-Bashforth 2 steps solver

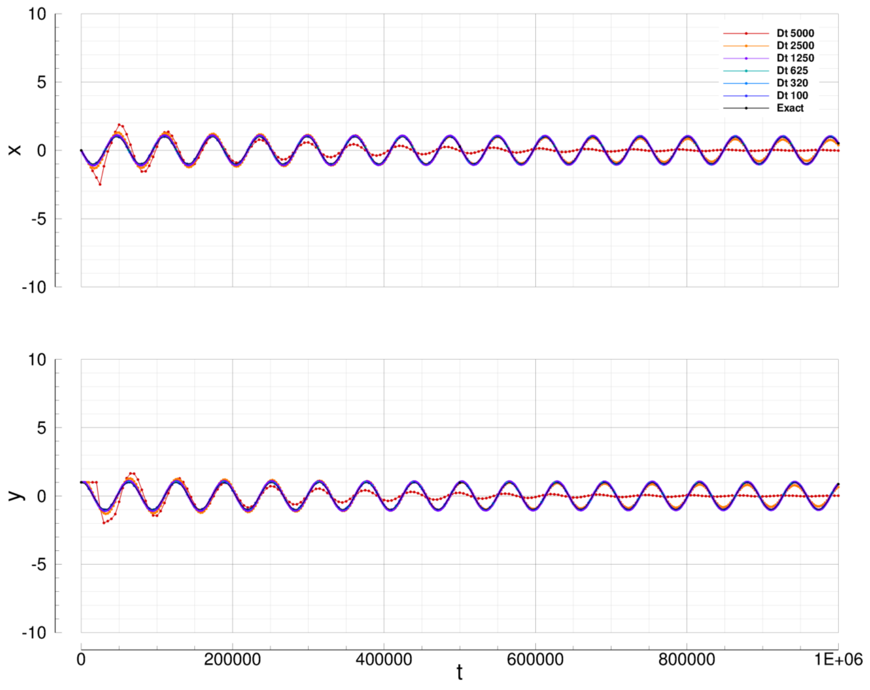

Figure 2: Oscillation equations solutions computed by means of Adams-Bashforth 3 steps solver

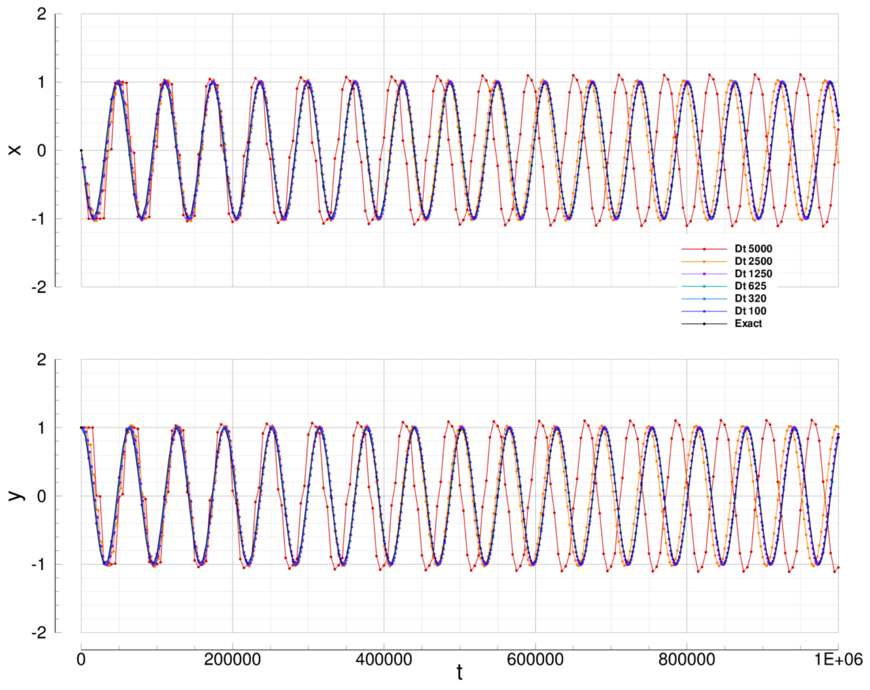

Figure 3: Oscillation equations solutions computed by means of leapfrog 2 steps solver

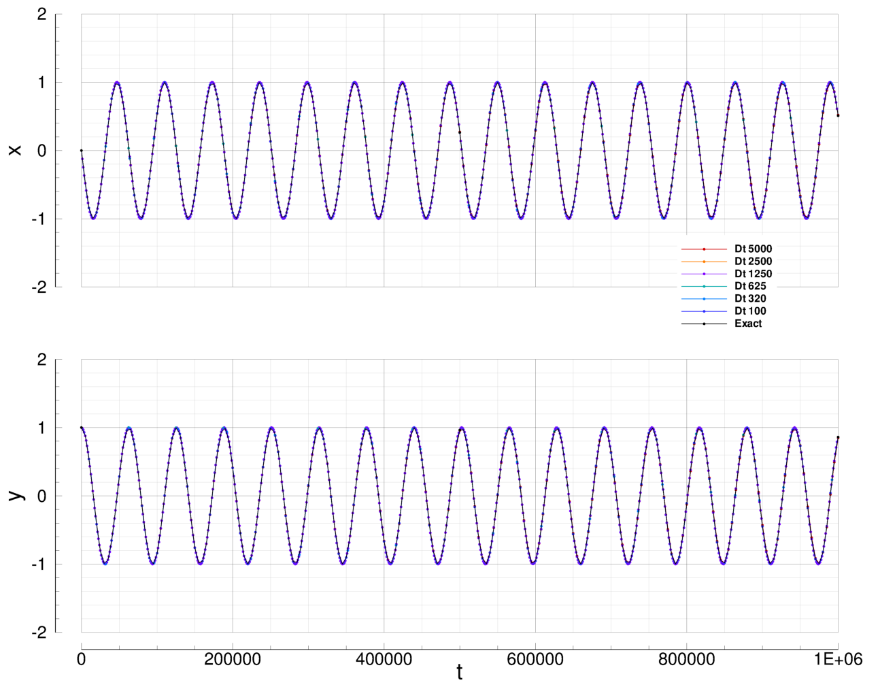

Figure 4: Oscillation equations solutions computed by means of low storage Runge-Kutta 5 stages solver

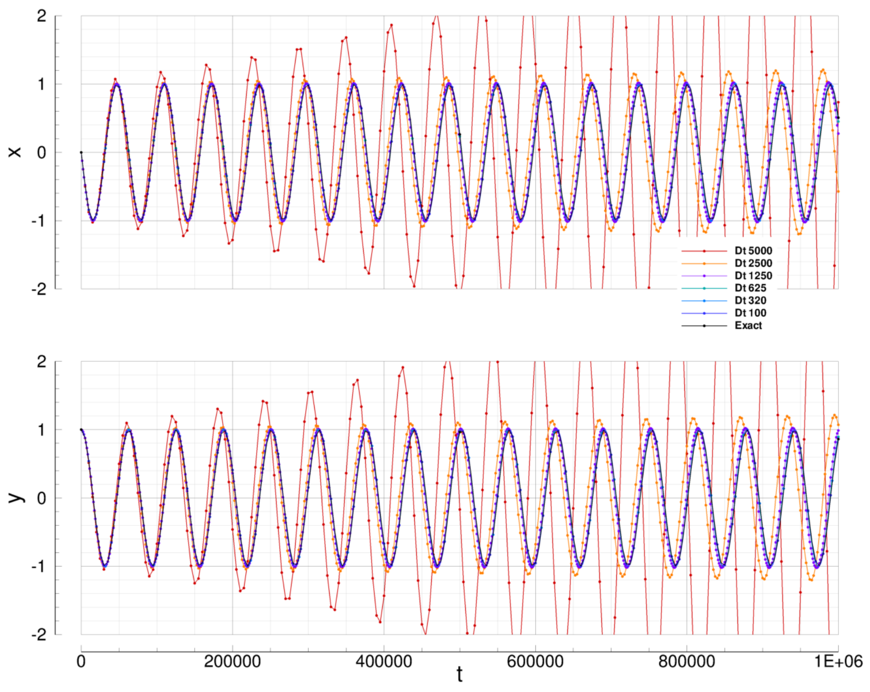

Figure 5: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta 2 stages solver

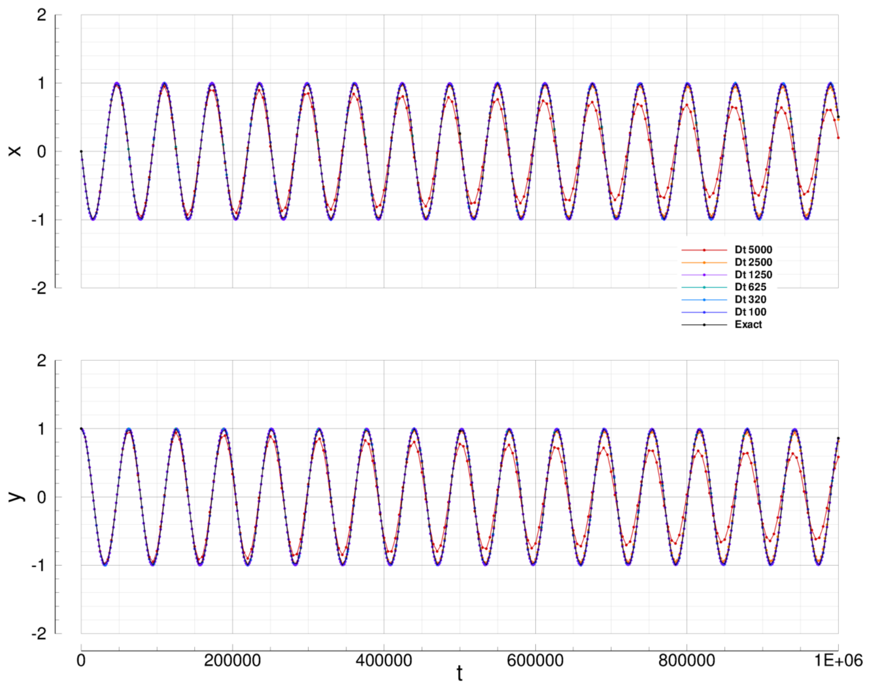

Figure 6: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta 3 stages solver

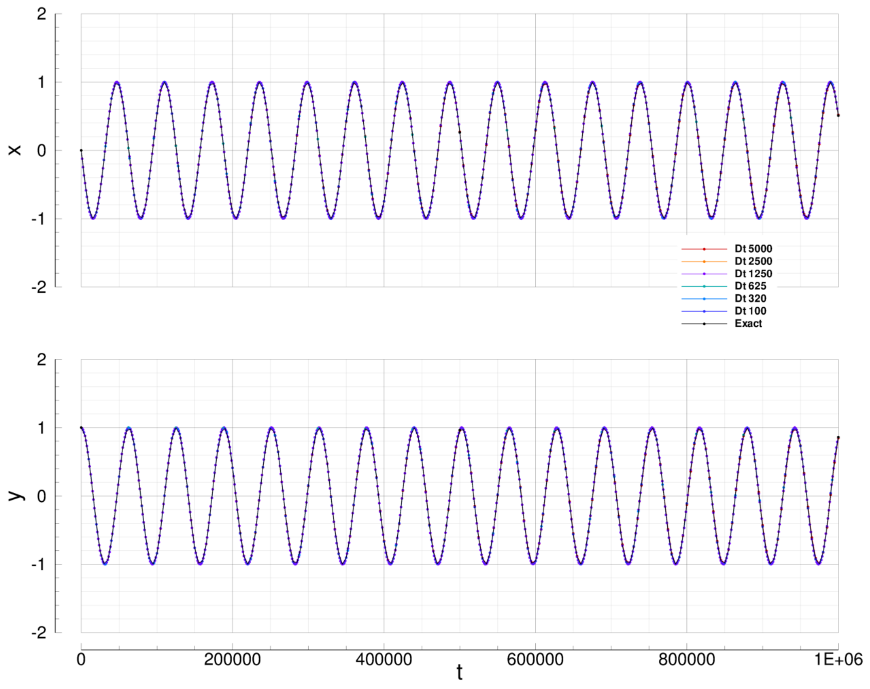

Figure 7: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta 5 stages solver

5. Concluding Remarks and Perspectives

To be written.

References